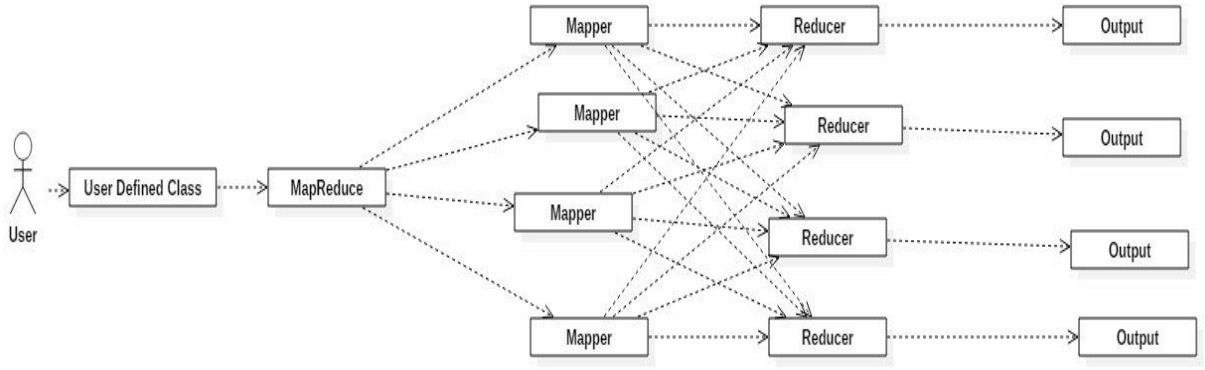
COMPSCI 532 SYSTEMS FOR DATA SCIENCE

- **Course:** COMPSCI 532 Systems for Data Science, Spring 2021
- **Instructor:** Hui Guan
- **Project**: MapReduce
- **Submission**: Mahidhar Kommineni, Sreerag Iyer, Sai Sameer Vennam

Overview

In this project, we have implemented the MapReduce library using Java. The user will be able to write arbitrary user defined Map and Reduce functions which will be compiled along with the source code to be able to perform an executable MapReduce job on a distributed system.

Flow of control:



As seen in the high level architecture above, users will execute the Map Reduce job. The map reduce job will then trigger multiple mapper processes which will invoke the user defined mapper functions on their respective partitions. The reducer processes are triggered after all the mapper processes have completed and the output of each reducer is written to a separate file.

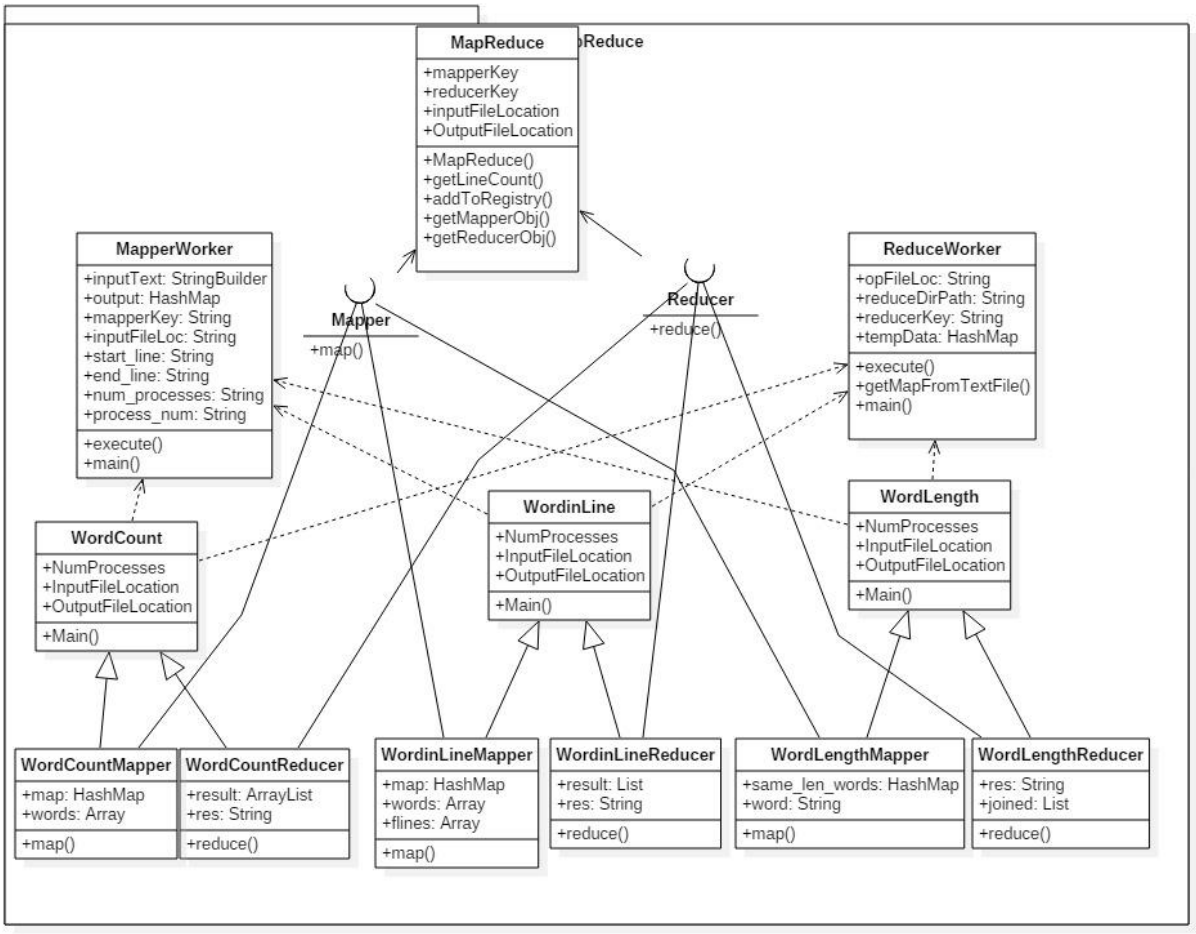
How it works and design considerations:

The master starts each mapper as a new process. Initially, an RMI registry will be started to facilitate communication between the master and the workers. Specifically, the RMI registry will be used by the workers to access the user defined map and reduce objects. The master (src/MapReduce) adds the user defined mapper and reducer objects to the RMI registry. The mapper launches N mapper processes. Each mapper receives an id, number of processes, location of the input file, the starting and ending line numbers of the text file it has to work on. Upon completion of all mapper processes, the master starts N reducer processes that write to N output files.

Distributing the output among intermediate files:

There are N mappers and N reducers. Each mapper produces N intermediate files, corresponding to each of the N reducers (i.e., if there are M mappers and R reducers, each mapper would produce R output files). For example, the first mapper will create a directory "mapper_0" with N files - "reducer_0.txt", "reducer_1.txt"...to "reducer_N-1.txt". Reducer N will then read "reducer_N-1.txt" from each of the directories "mapper_0", "mapper_1"...to "mapper_N". This design has the advantage of not requiring the master to explicitly assign intermediate files to reducers. The master only needs to give the locations of the N remote directories "mapper_0", "mapper_1" (and so on) that contain the intermediate files to the reducers. The tradeoff for this is that a total of N*N intermediate files will be formed.

MapReduce UML Class Diagram:



Input parameters to the library

Apart from the user defined map and reduce functions, the user also has to pass certain parameters to the library as input-

- Number of processes (N): This will indicate the N mappers and N reducers
- Input file location: This specifies where the text file on which the MapReduce operation has to be run is present.
- Output file location: This parameter specifies the directory where the N output files from the MapReduce operation will be located.
- Object of the user defined mapper class which will be added to an RMI registry so Master and mapper workers can access it
- Object of the user defined reducer class which will be added to an RMI registry so Master and reducer workers can access it
- Timeout for each worker: If this time limit is exceeded, then the worker stops. This is to mitigate the effect of stragglers.

Function signatures of the map and reduce functions are as follows:

```
HashMap<String,List<String>> map(String,String)throws RemoteException

List<String> reduce(String,List<String>) throws RemoteException
```

Sorting of keys

The mapper computes which intermediate file to write a particular key to by hashing that key to an integer from 0 to N-1. This ensures that all mappers will write to the same intermediate file in their particular remote directory. For example, say mapper 2 and mapper 3 encounter a key “hamlet”, which gets hashed to the number 1 (we use Java’s inbuilt String.hashCode()%N for this). Since this key got hashed to 1, mapper 2 will write it to “mapper_2/reducer_1.txt” and mapper 3 will write it to “mapper_3/reducer1.txt”. So the reduce operation on all occurrences of the key “hamlet” will be performed by reducer 1. Hence, no shuffling of keys among reducers is required.

Fault tolerance

The following scenarios are considered for fault-

- Worker failure: one particular process got killed and the worker was unable to complete the task.
- Stragglers: One particular worker takes too long to complete.

We simulate fault by randomly selecting one of the N workers (one mapper and one reducer) to be faulty. The master deals with this crash by restarting the failed worker. To mitigate the effect of stragglers, the user has an option to specify a time limit for each worker. By default, this is set to 6 seconds. If the time limit is exceeded, the worker (mapper or reducer) is stopped.

Test Cases

We have taken a total of three user defined test cases upon which we will perform MapReduce jobs. The test cases which contain the business logic would be executed when we run the MapReduce source code. The user defined test cases are WordCount, WordLength and WordinLine. For each of the user defined test cases,there are three classes which are the test class with the main method which runs the entire logic of the test case by invoking the user defined Mapper and Reducer functions. The user defined mapper functions are invoked to run on each row of the partition assigned to it.The output of this user defined map function is then stored on an intermediate file present on the local file system. This is in the form of a key and a list of values which are associated with each key. After all of the user defined mapper functions are done executing, the work of the reducer begins. Each user defined reducer function handles the information belonging to one key only and then it writes its output to a seperate file. The final output of the MapReduce job would be the grouping of all of the outputs from the reducer.

WordCount

- The test case WordCount gives the number of occurrences of each word present in the file 'hamlet.txt'.
- There are a total of three classes that are present which are WordCount, WordCountMapper and WordCountReducer.
- The WordCount class builds on the MapReduce and MapReduce specification classes. In this class, the number of processes which are two, the input file location as well as the location where the output must be stored are specified. The mapperkey as well as the reducerkey are also specified.
- The WordCountMapper class extends the WordCount class and implements the Mapper and Serializable interfaces.

- The WordCountMapper class is executed over a portion of the input file. It gets its input in the form of a key and value pair.
- In the WordCountMapper Class, the input is converted to lowercase and then split into words.
- A HashMap is used which has a Key of String Data Type and a value of List of Strings.
- Each time a word occurs in the input, a "1" is appended on to the value for that particular word.
- The output of the WordCountMapper class is a key which is each unique word in the input and a list of values which are the number of times that particular word occurred in the input.
- This output is then stored in a temporary intermediate file and then the WordCountReducer class takes the value corresponding to a key as input and the intermediate file is deleted.
- The WordCountReducer Class aggregates all of the values corresponding to a key and then outputs a single key value pair with word as the key and the count as the value

WordLength

- The test case WordLength gives all the length of all the words that are present in the file 'lorenipsum.txt'.
- There are a total of three classes that are present which are WordLength, WordLengthMapper and WordLengthReducer.
- The WordLength class builds on the MapReduce and MapReduce specification classes. In this class, the number of processes which are three, the input file location as well as the location where the output must be stored are specified.
- The WordLengthMapper class extends the WordLength class and implements the Mapper and Serializable interfaces.
- The WordLengthMapper takes in the input and removes all of the whitespace and punctuation marks. If it encounters white space or punctuation, it goes through the already appended string and checks if there have been any words that are equal in length to the string size.
- If the word that we are currently considering has an equal length as some of the previous words, then we would append this word to the previous list of words, if there is no word that has an equal length as this, then we create a new list and then append the word.
- The output of the WordLengthMapper is then stored in an intermediate file. The WordLengthReducer takes the input corresponding to a key as the input value. After the WordLengthReducer takes input, the intermediate file is deleted.
- The WordLengthReducer then formats all the values corresponding to a key and then stores it into the corresponding output file for the reducer.

WordinLine

- The test case WordinLine gives the line number that each word in the input file 'hamlet.txt' is present in.
- There are a total of three classes that are present which are WordinLine, WordinLineMapper and WordinLineReducer.
- The WordinLine class builds on the Mapper and MapReduce specification classes.In this class, the number of processes which are two, the input file location and the location where the output is to be stored are specified.
- The WordinLineMapper class extends the WordinLine class and implements the Mapper and Serializable interfaces.
- The WordinLineMapper class gets a portion of the input and start line number of the input that is given.
- The given input is split into different lines.
- The lines are again further split into words and then stored added on to the hashMap as key and then the line number is denoted by the variable line

How to run

The shell script to run all the test cases is present in the root directory ("run-test-cases.sh"). This will compile all the library code with test cases and then run them. The shell script will need to have permissions to run on the user's machine.

```
./run-test-cases.sh
```

To run the spark implementation - the spark implementation is present in the src/spark folder.

```
python3 filename.py
```

Outputs

The MapReduce library will output N files, based on the number of processes. The spark implementation provides one output file.

The output for the test cases are present in the "tes_cases_output" folder. The output for the spark implementation is present in the "spark_output" folder. Both are in the root directory.